Computer Science

Extending Programming By Demonstration With Hierarchical Event Histories

David S. Kosbie Brad A. Myers

May 1994
CMU-CS-94-156

Carnegie Mellon

Extending Programming By Demonstration With Hierarchical Event Histories

David S. Kosbie Brad A. Myers

May 1994
CMU-CS-94-156

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

To appear in the proceedings of *EWHCI '94: East-West Human Computer Interaction*, St. Petersburg, Russia, August 1994.
Also appears as Technical Report CMU-HCII-94-102.

This research was sponsored in part by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597. Support also came from the National Science Foundation under grant number IRI-9020089. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, ARPA, or the U.S. Government.

Keywords: Programming by Demonstration, Hierarchical Event Histories, Application Interface, User Interfaces, Intelligent Interfaces, End-User Programming

Abstract

Programming by Demonstration, or PBD, is an exciting and developing branch of HCI research. With PBD techniques, end-users can add functionality to their environments without programming in the conventional sense. Virtually all research into PBD, however, presumes that the event history is a linear sequence of user actions. This paper challenges that notion by introducing *Hierarchical Event Histories*, a new approach which represents some of the end-user's task structure directly in the event history. PBD systems can then take advantage of this structure to operate more correctly and in more situations. To assist programmers in generating structured histories, we also present *Hieractors*, a new model that provides a simple and clear syntax for describing arbitrary, high-level application behaviors.

1 Introduction

In the early days of computers, there was little distinction between the *programmer* and the *end-user*. Programs were designed to solve a particular task, and to be used exclusively by programmers. Today, this could hardly be less true. Programs such as word processors, spreadsheets, and databases are designed to be very general and apply to a large class of problems. Contemporary end-users, meanwhile, have little or no experience or even interest in programming.

These two trends have created a problem: generic software packages must be *customized* to suit end-users' specific needs, but many end-users have no means available to them to do such customization. This often results in end-users performing tedious, repetitive tasks that computers *could* have performed for them. For example, consider the simple task of using a word processor to insert a line number before each line in a large document. We posed this problem to a small sample of our colleagues, and they all came to the same conclusion: programming. Some considered Emacs macros, or Hypertalk scripts, or even Unix scripts. However, not one respondent knew how to perform this task *without programming*. In fact, in virtually all word processors, there is no other alternative. Thus, most end-users would have no choice but to painstakingly enter all the line numbers manually. Sadly, a large amount of human-computer interaction is exactly this sort of tedium.

1.1 Programming by Demonstration

These issues prompted research into *Programming by Demonstration*, or PBD, an exciting and developing branch of HCI. The basic goal of PBD is to allow end-users to customize their software by *demonstrating* the desired behavior. In the line numbering example, the user might type "1" on line 1, and "2" on line 2. From this, the PBD system should *infer* the line numbering task, and perhaps automatically complete the task for the user. Indeed, research systems like Eager [3] can already do this. Other systems have applied PBD to such domains as widget creation [18], graphical editing [15], and general-purpose programming [16].

The key advantage of PBD is that it allows end-users to specify programs *in the user interface*. They do not have to learn any special syntax or

programming constructs. In essence, they do not have to program in the conventional sense, yet they are able to customize their software to suit their particular needs.

1.2 Challenges to PBD

PBD is a technology of great promise. However, there are numerous problems yet to be solved before PBD will realize its full potential. These include:

- **User Intent**
  The primary concern of PBD is determining the *user's intent* in performing some actions. This requires *generalizing* the user's actions into a script which runs correctly under different circumstances. For example, say we have a word processor which has a `Style` menu and one of this menu's choices is `Bold`. Selecting `Bold` toggles the boldness of the selected text. We next demonstrate a script where we only select `Bold` from the `Style` menu. What should happen when we replay the script? This is unclear. We *might* have intended to record *setting* the text to bold. However, we also might have intended to record *toggling* the boldness of the text. Moreover, the difficulty of determining user intent grows quickly as the complexity of scripts increases.

- **Context**
  PBD systems often require access to the *context* in which a demonstration occurs. For example, if the user's intention was to set the text to bold, the inferred script should resemble the following:

  ```
  unless <the-selected-text-is-bold>
    select "Bold" from the "Style" menu
  ```

  The `unless` is necessary to prevent toggling when the selected text is already bold. To create this script, the PBD system must know whether the selected text was bold during the demonstration (*i.e.*, it must have accessed some context of the word processor). This poses several unsolved problems; specifically, how should the PBD system

  - determine the available context from an application?
  - access the context?
  - reason over the context?

- **Script Matching**
  Many behaviors are too complex to infer from a single demonstration. In this case, users must give *multiple demonstrations*, showing how the script runs in different situations. Conditionals are the most common case: as users can demonstrate only one branch at a time, conditionals require multiple demonstrations. This presents a problem: given two demonstrations of the same script, the PBD system must *match* the scripts, determining which steps are the same and which differ (and, ultimately, *why* they differ). Matching can be complicated because there can be several ways of accomplishing the same task, and users may be inconsistent across examples. For example, a desktop interface might support file deletion either by dragging the file icon to the trash icon, or first selecting the file icon and then selecting `Delete` from the `File` menu. While both methods satisfy the same high-level goal, few existing PBD systems could match them. This may generate a useless rule for selecting which method to use, which then may require more examples than are strictly necessary to learn the behavior.

- **Anticipation Feedback**
  Script mismatches can be reduced with *Anticipation Feedback*, as demonstrated in Eager [3]. With this approach, the PBD system encourages consistency across examples by providing feedback indicating what event the PBD system anticipates will next occur. For example, if the PBD system anticipates that the user will select a certain button, it may highlight the button in green. The user can then perform the action, or tell the PBD system to do it. In any case, if selecting the button is a reasonable alternative, the user is more likely to do so.

  Developers wishing to include Anticipation Feedback in their applications must address the *reverse-mapping* problem: if the PBD system records events at a high level (as most do), these high-level events must be mapped back into widget-level events for anticipation. To accomplish this, the PBD system must first be aware of the possible mappings. Second, it must *choose* one, probably the same one the user last chose. Indeed, Eager includes special code to do this. The challenge is to provide this to PBD systems in a general manner.

- **Invocation**
  In [14], we propose that PBD systems should allow users to not only demonstrate their programs but to also demonstrate *when to invoke*

those programs. Furthermore, there should not be restrictions on the kinds of events which invoke programs. Most PBD systems support a small, fixed selection of invoking events, such as clicking on certain icons or choosing certain menu items. This limits the utility of the PBD system, however. For example, say that a user wishes to copy all files to a backup directory before they are deleted. The script which performs the copying is easy to demonstrate, but most systems could not invoke the script before each `Delete-File` event. Thus, expanding the invocation techniques extends PBD to solve problems it otherwise could not.

This is by no means an exhaustive list. Other issues include how to represent the inferred script, allow the user to edit the script, and recover from errors while running the script. A more complete discussion of these issues is in [5].

1.3 High-Level Event Histories

A major factor in the quality of a PBD system is the level at which events are recorded. PBD systems based on *device-level events* (*i.e.*, mouse and keyboard events) are very unreliable. For example, if a `Mouse-Down` event selected some object, replaying the same event would select the same object only if the object is uncovered, in the same location, and not selected. Indeed, the same `Mouse-Down` event might invoke other, possibly destructive behavior.

In response to these concerns, various notions of *high-level events* were developed. High-level events vary by system, but generally equate to user actions such as `Delete-File`, `Make-Bold`, and `Quit-Application`. In these systems, an application processes low-level events in the normal manner until it determines that a high-level event should be performed. This event is then passed to the PBD system, *where it is recorded*, then back to the application, where it is finally executed. Thus, PBD systems can ignore device-level events, and reason over high-level event histories. This produces scripts which are more correct, more efficient, and more understandable.

2 Hierarchical Event Histories

The same arguments that favor high-level events over device-level events, however, also favor even *higher*-level events. Moreover, there is an occasional need for low-level events, too. For example, consider the user of a word processor saving the current file under the name "foo". To do this, she first selects **Save** from the **File** menu. This generates a dialog box for specifying the filename. There is a default value—the current filename—so she enters **control-u** to delete the text. She then enters the new name. Finally, she clicks on the "OK" button. How should the event history depict this sequence?

One possibility is a single high-level event, namely **Save-File("foo")**. This has the virtues listed in the previous section on high-level events. Unfortunately, it is also limiting. The most compelling argument is based on *correctness*—occasionally, only device-level events accurately portray the user's intent. Say the user demonstrates making a backup copy by appending ".bak" to the current filename. If the current filename is "foo", the macro should save the backup as "foo.bak". To demonstrate this, the user brings up the dialog box from the previous example, but does not delete the filename. Instead, she appends to it by typing ".bak". Then she clicks on the "OK" button. How should this sequence appear in the history?

The corresponding high-level event is **Save-File("foo.bak")**. Replaying this when the current filename is "bar", however, would produce a file called "foo.bak", not "bar.bak". An advanced PBD system might fix this with *context* and *inferencing*, generalizing the event to:

```
Save-File(append *current-filename* ".bak")
```

However, the PBD system might require vast time and space resources to make this inference. Even worse, it might fail to infer this at all. Notice, however, that the net effect of the device-level events is equivalent to the generalized **Save-File** event. Thus, if device-level events are in the event history, they can be replayed directly, and *no inferencing is necessary!*

Including low-level events in the history has additional benefits for PBD and other areas as well. In the file-saving example, events such as **Open-Dialog**, **Set-String**, and **Close-Dialog** can be applied to:

- **Invocation:** A third-party vendor might provide a *macro-based help facility* for the word processor. This might include a window with

some text on how to specify the filename when saving a file. The help facility would include a macro which displays this window upon the `Open-Dialog` event, and another macro which hides the window upon the `Close-Dialog` event. Similarly, another vendor might supply an *automatic spell-checker* which is invoked by the `Set-String` event.

- **Undo:** Suppose the user errantly typed `control-u` and deleted the current filename. In most systems, she must then retype the entire filename or abort the save operation. Including low-level events in the event stream, however, allows her to Undo the errant `Key-Press`.

- **Anticipation feedback:** Say that the PBD system correctly anticipates that the user will next issue a `Save-File` event. How should this be conveyed to the user? This is the reverse-mapping problem mentioned above. If the widget-level events are in the event history, the PBD system can iteratively anticipate those.

Thus, many different levels of events should be included in the history. However, it is semantically incorrect to include multiple event levels in a *linear* event stream. That is, the history cannot be flat, as in Figure 1. If

```
Mouse-Down
Mouse-Move
    ...
Mouse-Up
Select-Menu-Item('Save' 'File')
Open-Dialog('Save File')
Key-Press(".")
Key-Press("b")
Key-Press("a")
Key-Press("k")
Set-String('foo.bak')
Mouse-Down
Mouse-Up
Select-Button('OK')
Close-Dialog('Save File')
Save-File('foo.bak')
```

Figure 1: A flat history comprised of multiple levels of events.

this sequence were played back, the device-level events would generate extra instances of each high-level event. This would result in *three* `Close-Dialog` events, for example. Thus, if multiple event levels are in the event stream, the event stream itself cannot be linear—it must reflect the actual hierarchy of events. In this example, it must be structured as in Figure 2.

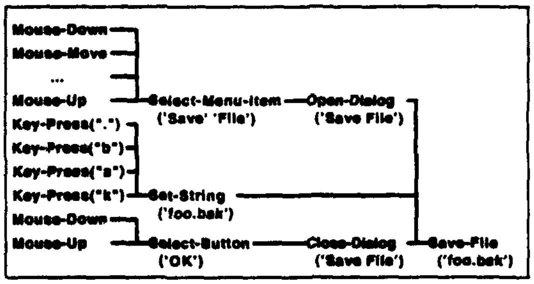

Figure 2: The same history as in Figure 1, but here we show the hierarchical structure.

This argument is the basis for *Hierarchical Event Histories*. We propose that applications should be structured so that they generate histories such as the one above. This will allow PBD systems, Undo facilities, and other history mechanisms to operate more correctly and in more situations.

3 The Hieractors Model

For hierarchical event histories to be feasible, there must be a simple way for application designers to *generate* them in the first place. We attacked this problem in three ways. First, we hoped to *infer* the structure based on the read-write patterns of the event handlers. This approach is ideal from the programmer's perspective, since it would work with unmodified code. Unfortunately, while we could infer the *partial sequential order* of events over time, read-write patterns alone do not provide enough information to completely determine the hierarchical structure.

We then tried the opposite approach, requiring the programmer to *explicitly construct* the hierarchy. To that end, we provided functions such as `Create-Event`, `Add-Child`, and `Set-Parent`. This approach worked fine for simple cases, but did not scale well. As we attacked harder problems, we had to repeatedly add new history-manipulation functions. These became so unwieldy that it was impractical to continue in this direction.

From these efforts we concluded that:

- programmers should be responsible for generating the event hierarchy; and
- there should be some architectural support to simplify this process.

We satisfied these criteria by developing *Hieractors*, a new model for expressing high-level behaviors. Hieractors (from *Hierarchical Interactors*) are inspired by the Interactors model [19]. Interactors are effective in describing widget-level interface behaviors. Hieractors generalize this model to provide a simple and clear syntax for describing arbitrary, high-level application behaviors.

The basic idea behind hieractors is that most application behaviors are naturally defined in terms of the events which start, run, stop, and abort them. For example, consider the behavior of selecting a button in a graphical interface. The hieractor providing this behavior would start on a **Mouse-Down** in the button, run on **Mouse-Moves**, and stop on a **Mouse-Up** in the button (aborting on other **Mouse-Ups**). This behavior is supplied by the following code fragment:

```
(create-instance 'Button-Behavior *hieractor*
  (:result-type  'Select-Button)
  (:start-when   'Mouse-Down)
  (:running-when 'Mouse-Move)
  (:stop-when    '(Mouse-Up :where in-original-button?))
  (:abort-when   '(Mouse-Up :where outside-original-button?))
  ...)
```

When a hieractor completes, it issues a higher-level event (in this example, a `Select-Button` event). The children of this event are precisely the events which triggered the start, running, and stop actions. This leads to the simple, but hierarchical, history seen in Figure 3.

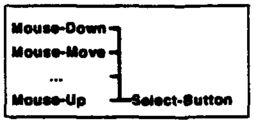

Figure 3: A simple hierarchical history for selecting a button.

High-level events can also start, run, stop, or abort even *higher*-level behaviors. This leads to more complex hierarchical histories, such as the history depicted in the file saving example above. To illustrate, the `Save-File` event from that example could have been generated by the following:

```
(create-instance 'File-Saver *hieractor*
  (:result-type  'Save-File)
  (:start-when   '(Open-Dialog :where save-file-dialog?))
  (:running-when 'Set-String)
  (:stop-when    '(Close-Dialog :where save-file-dialog?))
  (:abort-when   '(Abort-Dialog :where save-file-dialog?))
  ...)
```

Expressing the common high-level application behaviors required some extensions to this model, the most important being:

- **Event Combinations**
  Some behaviors make a transition (*i.e.*, start, run, stop, or abort) after a *sequence* of events. For example, in a calculator, the `Add-Numbers` behavior starts on an `Enter-Number` followed by a `Select-Button` where the selected button's label is "+". Similarly, transitions can occur after *either* of two (or more) events.

- **Scoping**
  When a hieractor, such as the "file-saver" above, is defined, it is attached to an object called its *scope*. The hieractor can only observe events which pass through this scope. For widgets, the scope usually corresponds to the graphical objects making up the widget. Thus, mouse clicks over a scrollbar are not needlessly processed by, say, the menubar in the same window. Sometimes behaviors must observe events from multiple widgets, or events that are not even associated with widgets. These behaviors can have a scope of any window, any application, or the entire system (where they observe every event).

  Behaviors sometimes start in one scope, but run in another. For example, consider moving an object in a graphical editor. When idle, this behavior should wait for a `Mouse-Down` event *inside an object*. Once running, however, it must observe `Mouse-Move` events *anywhere in the window*. For this reason, behaviors can specify a separate *running scope*. For convenience, this can be specified relative to the initial scope. In the example, the running scope is `:window`, meaning whatever window the object is in.

- **Priorities**
  It is possible (and common) for two hieractors to claim the same event.

In fact, the *same* hieractor can make multiple claims on a single event. For example, consider editing a one-line text field. This starts on any **Key-Press** event, but it also runs over other **Key-Press** events, and stops when the user hits **Return**. This is specified as:

```
(create-instance 'Simple-Text-Editor *hieractor*
  (:start-when   'Key-Press)
  (:running-when 'Key-Press)
  (:stop-when    '(Key-Press :where return-key?))
  ...)
```

Say the user types "H", then "i", then **Return**. The "H" unambiguously starts the hieractor. The "i", however, can either be a running event or *another start event*. Which transition should be favored by the event dispatcher? One solution is to disallow hieractors from starting while running. This is unnecessary and restrictive, however. Instead, transitions have integer *priorities*, where the larger priority is favored (and ties are decided randomly). Also, priorities can be *absolute*, or *relative* to the start priority. By default, abort events have the highest priority, followed by stop events, running events, and finally start events. Thus, for example, the text editor would process the "i" as a running event.

A more complete discussion of Hieractors is given in [13]. Note, however, that we have satisfied the design criteria for Hieractors. First, the programmer, not the PBD system, defines the structure of the application. Second, Hieractors provide significant architectural support to assist in this task. Our experiences indicate that programming with Hieractors requires about the same effort as conventional programming. However, Hieractors provide hierarchical event histories, with all their advantages.

4 Advantages of Hierarchical Event Histories

The key advantage to our model is that it represents some of the end-user's task structure directly in the event history. PBD systems can then take advantage of this structure to operate more correctly and in more situations. Referring back to the various challenges facing PBD systems, hierarchical event histories address many of these issues:

- **User Intent**
  Hierarchical event histories can aid in determining user intent because they expose more levels of user actions to the PBD system. Each level of the event hierarchy is typically a *specialization* of the level below it, corresponding to the effect the lower level events have *in the current context.* For example, consider when the user selects `Bold` from the `Style` menu. This would produce a `Toggle-Bold` event, which then produces a `Set-Bold` event in some contexts and a `Clear-Bold` event in others. Thus, the history contains *both* the toggle and setting (or clearing) behaviors, enabling the PBD system to offer both options to the end-user ***without making any inferences!***

- **Context**
  While our approach does not address the context problem directly, it does reduce the situations in which context is even necessary. Or, to rephrase this, hierarchical event histories extend the coverage of PBD systems which do not have access to application context. This is because high-level events ***implicitly*** include some context, as just described. Note that this is only a partial solution, however, as the user's intent may depend on context that *is not implicit in the hierarchy.* Even in these cases, however, the PBD system can select from the various levels to choose which should be generalized.

- **Script Matching**
  One cause of mismatched events in multiple scripts is when there are multiple ways to perform some action, as in the file deletion example from above. Hierarchical event histories can reduce script mismatches in many of these cases, as the mismatched low-level events may be children of easily matchable high-level events. For example, say the user first demonstrates deleting a file by dragging it to the trash, and later demonstrates the same step by selecting the file and selecting `Delete` from the `File` menu. While the low-level events are completely different, both actions will produce *the same high-level event*, namely `Delete-File`, thus making the matching a trivial task. While this does not solve the generalization problem (*i.e.*, what the *arguments* to the `Delete-File` should be), at least it advances the PBD system to that step.

- **Anticipation Feedback**
  Hierarchical event histories provide exactly the low-level support needed

to solve the reverse-mapping problem for Anticipation Feedback. This is because the recorded script *contains the widget-level events.* This enables the PBD system to anticipate an event above the widget level by iteratively anticipating the low-level events it comprises.

- **Invocation**
  PBD systems that allow arbitrary events to invoke user-defined programs can further benefit from hierarchical event histories. Including high-level events such as `Delete-File` in the event history allows programs to be invoked when these events occur. Moreover, including low-level events such as `Key-Press` and `Select-Button` in the event history supports the invocation techniques currently available to users. By exposing more levels of a user's task structure, our approach gives users more control over how and when their programs are invoked.

There are additional benefits to hierarchical event histories. For example, by allowing recorded scripts to be replayed at the highest semantically correct level, they can be *more efficient* than linear events. Also, while outside the scope of this paper, hierarchical event histories benefit other parts of HCI, such as Undo, Help, and Task Analysis.

5 Related Work

For people interested in learning more about Programming by Demonstration, [4] presents a thorough overview and history of the field and describes the current state-of-the-art. The crucial problem of determining user intent was first described in [9]. While many systems have made inroads on this problem, perhaps the most promising is Cima [17], a learning architecture being developed specifically for PBD systems. We are currently pursuing ways to integrate our work with the Cima environment.

While there are many user interface specification techniques (such as [10, 12, 8]), these do not address the nature of the event history. Approaches such as TAG [21] and GOMS [2] do consider the hierarchical task structure, but not how to *generate* such a history (*i.e.*, they are *analytical*, not *constructive*). A more hybrid approach is taken in Task-Oriented Parsing [11], which is somewhat constructive and hierarchical. It is based on context-free grammars, however, which are less powerful than event-based models, and

cannot describe some important user interface behaviors [8]. Moreover, their approach is not truly constructive because they provide "normal feedback" only for "meaningful tasks", and not "all [user] input actions."

The simpler high-level event model is supported by numerous systems. In particular, most model-based UIMS's, including MIKE [6], [illegible] [7], Humanoid [22], and others. We extended Garnet [20] because of [illegible] expertise with that model, and because the resulting Hieractors model is clear, concise, and efficient. It seems reasonable that other model-based UIMS's could be adapted to generate hierarchical event histories as well. Also, Apple Events [1] are a high-level event paradigm now employed by a large and growing vendor population. Because of this, we are considering converting Hieractors to operate over Apple Events.

6 Status and Future Work

The ideas presented here serve as the basis for Katie [13], an application environment which includes a Hieractors interpreter for the basic model with the extensions listed in this paper. Katie also includes two widget sets (a basic set and a more complicated Motif look-and-feel set), several small applications, and a larger database-type application. At this point, we have proven the viability of the Hieractors model for generating hierarchical event histories. The next phase of this research will focus on the graphical presentation and manipulation of the structured history, and the many applications of hierarchical event histories.

7 Acknowledgments

We would like to thank Allen Cypher, Francesmary Modugno, and James Landay for their help with this paper.

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597.

The views and conclusions contained in this document are those of the

authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

References

[1] Apple Computer, Inc. *Inside Macintosh Volume VI.* Addison-Wesley, Reading, MA, 1991.

[2] Stuart K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human-Computer Interaction.* Lawrence Erlbaum, Hillsdale, NJ, 1983.

[3] Allen Cypher. Eager: Programming repetative tasks by example. In *Human Factors in Computing Systems*, pages 33–40, New Orleans, April 1991. ACM SIGCHI.

[4] Allen Cypher, editor. *Watch What I Do: Programming by Demonstration.* MIT Press, Cambridge, MA, 1993.

[5] Allen Cypher, David S. Kosbie, and David Maulsby. Characterizing PBD systems. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 467–484. MIT Press, Cambridge, MA, 1993.

[6] Jr. Dan R. Olsen. Mike: The menu interaction kontrol environment. *ACM Transactions on Graphics*, 5(4):318–344, October 1986.

[7] James Foley, Won Chul Kim, Srdjan Kovacevic, and Kevin Murray. Defining interfaces at a high level of abstraction. *IEEE Software*, 6(1):25–32, January 1989.

[8] Mark Green. A survey of three dialog models. *ACM Transactions on Graphics*, 5(3):244–275, July 1986.

[9] Daniel C. Halbert. SmallStar: Programming by demonstration in the desktop metaphor. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 103–124. MIT Press, Cambridge, MA, 1993.

[10] H. Rex Hartson, Antonio C. Siochi, and Deborah Hix. The UAN: A user-oriented representation for direct manipulation interface designs. *ACM Transactions on Information Systems*, 8(3):181–203, July 1990.

[11] Heinz Ulrich Hoppe. A grammar-based approach to unifying task-oriented and system-oriented interface descriptions. In D. Ackermann and M.J. Tauber, editors, *Mental Models and Human-Computer Interaction 1*, pages 353–374. North-Holland, New York, 1990.

[12] Robert J.K. Jacob. A specification language for direct manipulation interfaces. *ACM Transactions on Graphics*, 5(4):283–317, October 1986.

[13] David S. Kosbie. Hierarchical event histories. PhD thesis. In preparation, 1994.

[14] David S. Kosbie and Brad A. Myers. PBD invocation techniques: A review and proposal. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 423–432. MIT Press, Cambridge, MA, 1993.

[15] David Kurlander and Steven Feiner. Editable graphical histories. In *Workshop on Visual Languages*, pages 127–134, Pittsburgh, October 1988. IEEE.

[16] Henry Lieberman. Tinker: A programming by demonstration system for beginning programmers. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 49–66. MIT Press, Cambridge, MA, 1993.

[17] David Maulsby. *Instructible Agents*. PhD thesis, University of Calgary, Calgary, Alberta, Canada, 1994. PhD thesis.

[18] Brad A. Myers. *Creating User Interfaces by Demonstration*. Academic Press, Boston, 1988.

[19] Brad A. Myers. A new model for handling input. *ACM Transactions on Information Systems*, 8(3):289–320, July 1990.

[20] Brad A. Myers et al. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer*, 23(11):71–85, November 1990.

[21] Franz Sciele and Thomas Green. HCI formalisms and cognitive psychology: The case of task-action grammar. In Michael Harrison and Harold Thimbleby, editors, *Formal Methods in Human-Computer Interaction*, pages 9–62. Cambridge University Press, Cambridge, 1990.

[22] Pedro Szekely, Ping Luo, and Robert Neches. Facilitating the exploration of interface design alternatives: The Humanoid model of interface design. In *Human Factors in Computing Systems*, pages 507–515, Monterrey, CA, May 1992. ACM SIGCHI.